

# PROGRAMMING EXPERT

## FAQ

**Q** What's the cheapest way to get started in programming?

**A** Many free computer languages lack support material. If you want access to lots of help and examples online, there are only two sensible options: Java or one of the .NET languages. Java is available for free download from <http://java.sun.com/j2se/1.5.0/download.jsp> and can be used to develop software to run on a range of platforms. If developing Windows applications is your main concern, Microsoft's .NET system is your best bet. The two major .NET languages, Visual Basic and C#, are tailored to Windows development and have free, cut-down Express versions from <http://msdn.microsoft.com/vstudio/express/default.aspx>.

**Mike James**

IT author, lecturer  
and programmer

[mike@computershopper.co.uk](mailto:mike@computershopper.co.uk)

# .NET inheritance

If you find the .NET system is lacking in some way, you can add controls that inherit properties, says Mike James



**A**lthough .NET is a fairly mature system, having reached version 2.0, it still lacks the richness of the non-.NET-based Visual Basic 6. This is a problem if you are familiar with VB6 or are trying to convert existing applications. Fortunately, the whole point of object-oriented .NET languages is that they are extensible. You can add features to .NET. Often this is surprisingly easy, as many of the 'missing' facilities aren't missing, just hiding below the surface. Here we will recreate the apparently missing DriveCombo box control using nothing but standard C# .NET.

VB6's DriveCombo box is a useful control because it provides the user with an easy way to select a drive from a drop-down list complete with icons that indicate the drive type. It also serves as a blueprint for constructing more sophisticated combo boxes that combine text and graphics. Before we look at how to build a DriveCombo box, we have to solve the problem of finding the appropriate icons for the drive types.

In this project we will learn how to use the extended system icons, create new classes that wrap existing Windows facilities, use the DrawItem to take over how a control displays and create a new control using inheritance.

## SHELL ICONS

Windows uses a wide range of standard icons, the location of which seem to be well hidden from the average application programmer. If you spend time searching the documentation, you probably won't find out much about how you can use these icons, but they aren't deliberately hidden. The .NET framework provides some access to standard system icons in the SystemIcons class. This returns an Icon object for any of the icons that you find in a message box or dialog box, but not the wider range of drive and folder icons. To find these, you need to look in the Shell32.dll file in the Windows\System32 directory. There is an ExtractAssociatedIcon method in the Icon class that will extract a single icon from a file, but it returns only the first icon found in the file. To pick an icon from the

Shell32.dll file, we have no real choice but to use an API call: ExtractIcon.

To make our additional shell icons look as much like the SystemIcon class as possible, we need to create a new class within the project called ShellIcons. As well as the standard classes, we also need to add:

```
using System.Drawing;
using System.Runtime.InteropServices;
```

There is no need to have more than one instance of the ShellIcons class so we can declare it to be static. A static class is instantiated automatically by the system and the resulting object has the same name as the class. We are going to add static properties to the class that return each of the icons we want to use. For example, to get the 'hard disk' icon, you would use something like:

```
Icon icon=ShellIcons.HardDisk;
```

To make the class static, add the static modifier:

```
static class ShellIcons
{
```

We also need some constants to help us locate icons in Shell32.dll. They are stored in order, with the first icon having index 0. We don't know of any documentation that gives the order of the icons in the file, but it is easy to find what you are looking for with a test program. The following constants give the position of the device icons in which we are specifically interested:

```
private const int Floppy = 6;
private const int Hard = 8;
private const int Net = 9;
private const int CD = 11;
private const int RAM = 12;
private const int Unknown = 53;
```

The API call is easy to translate to C#:

```
[DllImport("shell32.dll", EntryPoint =
"ExtractIcon")]#
extern static IntPtr ExtractIcon(#
IntPtr hInst,#
string lpszExeFileName,#
int nIconIndex);#
```

It returns a handle to the icon specified by `nIconIndex` in the file `lpszExeFileName`. The `hInst` parameter is supposed to be set to the application's handle, which is used by Windows to keep track of it. It is quite difficult to obtain the application instance handle. After spending some time implementing other API calls to discover this, a programming error revealed that the `ExtractIcon` API call doesn't seem to use it or need it. This is undocumented, so if you find omitting `hInst` causes problems with another version of Windows (we used Windows XP), you might have to implement the code needed to obtain the application handle.

### CONSTANT COMPANION

As well as using constants for the icon indexes, it makes good programming sense to use a constant for the location of the `Shell32.dll` file:

```
const string ShellIconsLib =#
@"C:\WINDOWS\System32\shell32.dll";#
```

It is worth adding a public method that returns the icon corresponding to any index:

```
static public Icon GetIcon(int
index)#
{#
    IntPtr Hicon = ExtractIcon(#
        IntPtr.Zero, ShellIconsLib,
        index);#
    Icon icon =
        Icon.FromHandle(Hicon);#
    return icon;#
}#
```

Notice the use of the `FromHandle` static method to create the icon object. Now we can add the properties to return the specific icons we are going to use. As there is no logical reason why the user should need to set these properties, we just define a `Get` procedure:

```
static public Icon FloppyDisk#
{#
    get { return GetIcon(Floppy); }#
}#
static public Icon HardDisk#
{#
    get { return GetIcon(Hard); }#
}#
static public Icon NetDisk#
{#
    get { return GetIcon(Net); }#
}#
static public Icon CDRom#
{#
    get { return GetIcon(CD); }#
}#
static public Icon RamDisk#
{#
    get { return GetIcon(RAM); }#
}#
```

```
#
static public Icon UnknownDisk#
{#
    get { return GetIcon(Unknown); }#
}#
```

You can add further properties to return other icons; the only problem is thinking up a name for some of them.

### BASIC COMBOBOX

The `ComboBox` control is easy to use. You place it on a form and add to its `Items` collection anything that you want to appear in its drop-down list. You can add general objects to the `Items` collection and the `ComboBox` will display their `Text` properties; this is an important detail, as we will see. We want to create a new control based on the `ComboBox`, but it is easier to modify a standard `ComboBox` first before turning it into a new control.

Start a new project and add the `ShellIcons` class and a `ComboBox`. First we need to populate the `ComboBox` with a list of available drives. This can be done in the `Form Load` event handler:

```
private void Form1_Load(object
sender, EventArgs e)#
{#
    DriveInfo[] Drives =
        DriveInfo.GetDrives();#
    foreach (DriveInfo Drive in
        Drives)#
    {#
        comboBox1.Items.Add(Drive);#
    }#
    comboBox1.SelectedIndex = 0;#
}#
```

The `DriveInfo` class can tell you everything you need to know about a drive. Its static `GetDrives` method returns an array of `DriveInfo` objects, one for each drive in the system. Once we have this array, we can step through it, adding each `DriveInfo` object to the `ComboBox`'s `Items` collection. When the `ComboBox` displays, it uses each `DriveInfo` object's `Text` property for the list. To use the `DriveInfo` class we need to add:

```
using System.IO;#
```

If you run this program as it stands, you will get a `ComboBox` preloaded with drive letters. This could be all you need, but it looks unimpressive. Drive icons would improve its usability.

### OWNERDRAW COMBOBOX

To add icons to the drive list, we must modify the way the `ComboBox` is drawn. The designers of the .NET Framework have done their best to make such modifications easy. If the object's `DrawMode` property is set to `OwnerDrawFixed` or `OwnerDrawVariable`, rather than draw the

items in the drop-down list itself, the `ComboBox` instead raises the `DrawItem` event. The `DrawItem` event handler is passed all the information and objects needed to draw the control. This allows you to create your own drawing routine and create whichever style of control you like.

There is a small complication, though. If you set `DrawMode` to `OwnerDrawVariable`, the `ComboBox` raises a `MeasureItem` event just before the `DrawItem` event. This allows you to tell the `ComboBox` how much space you need to draw the next item. In this case the icons are the same size, so we can use `OwnerDrawFixed` and dispense with the `MeasureItem` event handler; the `ComboBox` will work out how much space there is to draw an item.

There is one small, final detail before we implement the `DrawItem` event handler. The default mode of operation of a `ComboBox` is to allow the user to edit the entry or select an item. For the user to edit the entry, it cannot be a graphic. It has to be text. We have to make the `ComboBox` prevent the user editing or entering items by setting its `DropDownStyle` equal to `DropDownList`.

Now we are ready to implement the drawing routine. In the `Form Load` event handler, add the following lines to customise the `ComboBox`:

```
comboBox1.DropDownStyle =#
    ComboBoxStyle.DropDownList;#
comboBox1.DrawMode =#
    DrawMode.OwnerDrawFixed;#
```

If you run the program, the result will be a blank `ComboBox`, because it doesn't draw itself and neither do you. The easiest way to add the `DrawItem` event handler is to click on the `ComboBox` in the `Form Designer`, click on the event icon in the `Properties` window and double-click on the `DrawItem` event. This generates the template for an event handler in the code:

```
private void comboBox1_DrawItem(#
    object sender,#
    DrawItemEventArgs e)#
{#
```

The `DrawItemEventArgs` object passed to the routine contains a great deal of information concerning the drawing of the item. If there is no item selected – the display area should be left blank – the `Index` property will be equal to -1 and we do nothing:

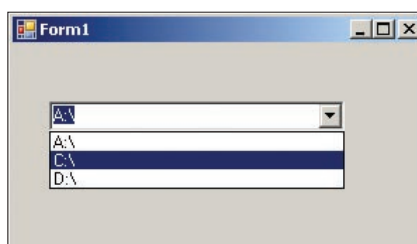
```
if (e.Index == -1) return;#
```

If an item has been selected, the `Index` will contain its position in the list. To begin drawing it, we have to acquire the right icon for the item's drive type. The `Index` is first used to obtain the `DriveInfo` object that we have to draw:

```
DriveInfo drive = (DriveInfo)#
    comboBox1.Items[e.Index];#
```

We use a cast to convert the object to a `DriveInfo` object. This is necessary because the `Items` collection is a collection of general objects and we need to tell the compiler what the object retrieved from the `comboBox` is.

Now we have the `DriveInfo` object, we can use its `DriveType` to retrieve the appropriate icon using the `ShellIcons` class:



▲ A plain drive list

## MICROSOFT PRESS Visual Basic 2005 Express Edition: Build a Program Now!

★★★★

£8.78

Visual Basic was the perfect beginner's language. You could avoid all the difficult elements until you were happy you understood the basics. Visual Basic 2005 is a great improvement on the original VB .NET.

It is easier to use, but it still isn't as user-friendly as the original, classic VB. Patrice Pelland's book is a re-write of his C# book and it fails miserably to take into account any of the problems that beginners have. It dives straight in with the complexities of object-oriented programming, and unless you have a strong constitution you're going to have problems.

If you have had some previous experience of Visual Basic, you might find the book helpful. It has the advantages of being cheap and including a copy of VB Express, together with all the examples from the book on a CD bound into the back. It's not a good book, and it certainly isn't a good introduction for the beginner, but the CD alone is worth £5.



### SUMMARY

- ✓ Copy of VB Express included
- ✗ Tricky for beginners

SUPPLIER [www.amazon.co.uk](http://www.amazon.co.uk)  
ISBN 0735622132  
PAGES 200

SAMS

## The Black Art of Video Game Console Design

★★★★★

£28

This is an amazing book and highly recommended, but only if you know what you're letting yourself in for. Andre LaMothe's book is not just another game-programming book. The focus is squarely on console design, and the book deals with hardware from the first page.

The initial chapters cover the physics that you need to understand electronics, including one of the best explanations of how a transistor works that we've ever read. We're not sure what a complete beginner would make of it all, as the author has a sophisticated view of the way things work. But if you are well-grounded in software, this could be the book you've been waiting for. It's great fun and explains things that you might have been wondering about for years. It goes from basic electricity to microprocessors and designing and building the hardware and software in an old-style games console.

You probably won't produce anything real or revolutionary through reading this, but if you're interested in the theory of how games consoles work, it's fascinating.



### SUMMARY

- ✓ Great explanations
- ✗ Nothing for beginners

SUPPLIER [www.amazon.co.uk](http://www.amazon.co.uk)  
ISBN 0672328208  
PAGES 840

```
Icon icon;
switch (drive.DriveType)
{
case (DriveType.Fixed):
    icon = ShellIcons.HardDisk;
    break;
case (DriveType.CDRom):
    icon = ShellIcons.CDRom;
    break;
case (DriveType.Network):
    icon = ShellIcons.NetDisk;
    break;
case (DriveType.Removable):
    icon = ShellIcons.FloppyDisk;
    break;
default:
    icon = ShellIcons.UnknownDisk;
    break;
}
```

The switch simply sets icons to the appropriate icon using ShellIcons. Seeing code like this, you might think there should be a better way of doing it. But the fact of the matter is there is no direct connection between DriveType and the specification for the icon, so you need to handle each case in turn. Sometimes you have to write lots of lines of code.

### SHAPING UP

Now we are ready to do the necessary drawing. The Bounds property of the DrawItemEventArgs, itself a Rectangle object, gives the coordinates of the area that we can draw in. We need to create a Rectangle object that specifies exactly where the icon is to be drawn from the Bounds Rectangle. We need a small square box (icons are always square) at the far left of the Bounds and filling the available height:

```
Rectangle iconbox = new Rectangle(
```

```
e.Bounds.X, e.Bounds.Y,
e.Bounds.Height, e.Bounds.Height);
```

You could add an offset to indent the icon if you wanted. Now we can draw the background and the icon in the area specified by the IconBox Rectangle using the Graphic object passed to the event handler:

```
e.DrawBackground();
e.Graphics.DrawIcon(icon, iconbox);
icon.Dispose();
```

Finally, we have to draw the text that would otherwise be displayed automatically by the ComboBox. When you take over drawing a control, you take over everything. This is quite easy to do, as we can obtain the drive's name and its volume label – if one exists – from the DriveInfo object:

```
string text = drive.Name;
if (drive.IsReady)
{
    if (drive.VolumeLabel != "")
    {
        text = text + " (" +
            drive.VolumeLabel + ")";
    }
}
```

The text can be drawn on to the Graphics object using the DrawString method. However, we have to specify the font, colour and location of the top left-hand corner of the text. All this information can be derived from the DrawItemEventArgs object:

```
e.Graphics.DrawString(text,
e.Font,
new SolidBrush(e.ForeColor),
```

```
e.Bounds.X+iconbox.Width+1,
e.Bounds.Y);
```

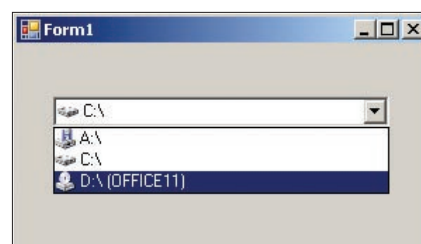
At this point, we could declare our work done. But the Status property tells us when the user has the mouse hovering over a selection and we can respond to this by drawing a select frame:

```
if (e.State == DrawItemState.Focus)
{
    e.DrawFocusRectangle();
}
```

This completes our modifications. If you run the example code on your cover disc, you will see a list of the available drives with icons. You can select a drive, and the control behaves in every other respect like a ComboBox.

### A DRIVE COMBO CONTROL

The code described above does the job but still needs work. The first problem is that it is fragile. You will hear this term often in modern programming theory as something to avoid. In this case, we have a ComboBox control, but its behaviour is controlled by code that lives on the form. This code is exposed to possible change as the rest of the form is created. Also, our new ComboBox isn't easy to reuse. You would have to



▲ A Drive ComboBox complete with icons



copy the code in the form to any other form that needed a Drive ComboBox.

A far better approach is to create a new control with all its workings built in. The custom control should be designed so you can place it on a new form using the toolbox in the normal way. This should be possible with an object-oriented language, but Microsoft's documentation on how to achieve this is confusing.

Start a new Windows form-based project and add to it a new class called DirCombo. This generates the following code:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WindowsApplication1
{
    class DirCombo
    {
    }
}
```

Now add:

```
using System.Windows.Forms;
```

Change the class declaration to:

```
class DirCombo:ComboBox
```

This line causes our new class to inherit from the ComboBox class. Inheritance is a feature of object-oriented programming that you don't use every day but when you do need it, you can't imagine any other way of achieving the same result in a reasonable amount of time. In this case, the meaning of inheritance is clear; the new DirCombo class has all the properties, methods and events of the ComboBox class. This makes it an exact copy of the ComboBox. If you compile or build the project, you will discover that the DirCombo control appears, with a default icon, in the toolbox. You can select it and place it on the form in the usual way to create a perfectly standard ComboBox.

However, we don't want an exact copy of the ComboBox class. The really useful thing with inheritance is that you can extend and modify

the inherited properties, methods and events. We want to give the new control the OwnerDraw behaviour that we have already seen.

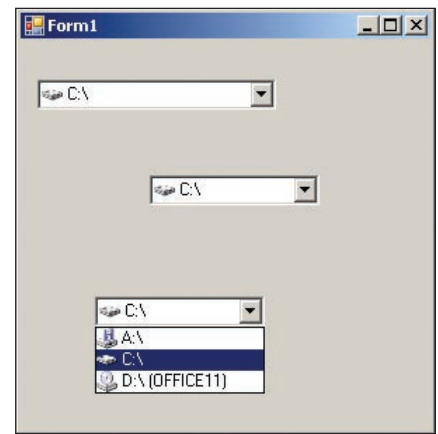
The new control needs access to the ShellIcon class, which can be done in several ways. The simplest way of keeping them together is to copy and paste the entire code of the ShellIcon class into the DirCombo file. You also have to add:

```
using System.Drawing;
using System.Runtime.InteropServices;
```

We need to perform all the initialisation that we did in the Form Load event handler in the new control's constructor:

```
public DirCombo()
{
    if (LicenseManager.UsageMode !=
        LicenseUsageMode.DesignTime)
    {
        DriveInfo[] Drives =
            DriveInfo.GetDrives();
        foreach (DriveInfo Drive in
            Drives)
        {
            base.Items.Add(Drive);
        }
        base.SelectedIndex = 1;
        base.DrawMode = DrawMode.Owner
            DrawFixed;
        base.DropDownStyle =
            ComboBoxStyle.DropDownList;
    }
}
```

The only real difference is that where before we referred to comboBox1, the specific instance of the class that was placed on the form, we now use the keyword base to refer to the class from which we inherit all the properties and methods. We also need to avoid initialising the control in design mode. This is the reason for the if statement, which is the standard way of discovering the current mode. Use this same technique to prevent your control doing anything inappropriate at design time. To make all this work, we also need:



▲ As many DriveCombo controls as you need

```
using System.IO;
using System.ComponentModel;
```

As this is the default constructor, the system automatically calls the base class constructor for you so the ComboBox class is initialised correctly before you start using it.

## MAIN EVENT

The most complicated task is to replace the ComboBox's DrawItem event. We don't want to allow this control to be customised, so we have to intercept the DrawItem event and run all the code that was in the event handler in the form (the code that gets and draws the icons and text). You might think you need to handle the event, but it is easier to override by replacing the ComboBox method that raises the event in the first place with our own version. When a ComboBox object wants to raise a DrawItem event, it calls its OnDrawItem method, which calls all the methods that have subscribed to the event as event handlers. We can override the ComboBox OnDrawItem method with our own code by adding to the new control class:

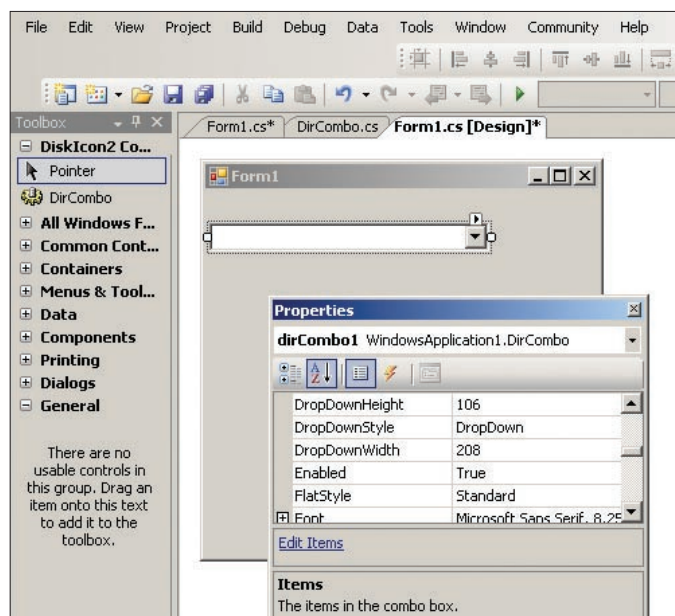
```
protected override void
    OnDrawItem(DrawItemEventArgs e)
{
}
```

If you run the program, you'll see the DirCombo box on the form works, but nothing is drawn. The new OnDrawItem method doesn't do anything. To make it draw the icon and text, we need to copy everything that was in the comboBox1\_DrawItem event handler associated with the form. As the only line of code that references comboBox1 is the one that gets the DriveInfo item, this is the only line that needs changing to refer to base instead:

```
DriveInfo drive = (DriveInfo)
    base.Items[e.Index];
```

Now if you run the project, you will see your DriveCombo control complete with Icons as before. The advantage is that now none of the code used to implement the control is on the form and you can create as many instances of the DriveCombo control as you need, simply by selecting it from the toolbox.

There are a few things left to do to tidy up the new control; a new toolbox icon would be nice, for example. But we have something that works. The complete code for both versions of the DirCombo control are on the cover disc. ☐



◀ The new DirCombo control works exactly like a ComboBox